intel®

APPLICATION NOTE

AP-341

September 1992

Designing an Updatable BIOS Using FLASH Memory

BRIAN DIPERT
DON VERNER
MCD MARKETING APPLICATIONS

Order Number: 292077-004

Designing an Updatable BIOS Using Flash Memory

CONTENTS

CONTENTS

1.0 INTRODUCTION

As PC computing platforms increase in complexity, so does the associated BIOS code. Sophisticated hardware and BIOS software increase the potential for revisions. Time-to-market goals require faster completion of designs from conception to production, leaving less time for new-peripheral BIOS support. As an example, many 80286-based PC/ATs lack BIOS support for $3\frac{1}{2}''$ floppy drives! Once a computer is out the door, code revisions are far more difficult and costly. Code revisions with EPROM require either a service call or sending EPROMs to the end user, assuming nothing else goes wrong in the process. The alternative to BIOS update is a prematurely obsolete system, unable to support new industry standards and peripheral systems.

Flash memory offers the same nonvolatile storage as EPROM, but additionally offers in-system write capability. Using Intel's 28F001BX for BIOS storage, code updates are done quickly in the factory during test and debug, while allowing cost-effective field updates to end users via floppy disks or modem BBS.

This application note describes various methods of implementing a flash memory BIOS using the 28F001BX. Design targets are both laptop and desktop systems. The primary emphasis is on application of flash memory for BIOS and ROM executable software applications. Detailed 28F001BX information is covered in the datasheet, available through your local sales office.

2.0 FLASH MEMORY

This section provides a brief overview of Intel's Flash Memory and in particular, Intel's 28F001BX blocked flash memory family. It covers the following:

- Flash memory's EPROM roots
- Program and Erase Automation
- Blocked Architecture
- Deep Powerdown Mode
- Pinouts, physical layout and upgrade for different packages
- V_{PP} specifications

Major features of the 28F001BX are in-system write, selective block erase, program/erase automation, SRAM-like command interface, deep powerdown capability, fixed V_{CC} and V_{PP} supplies and hardware lockout protection.

2.1 EPROM Roots; Review of Flash Process vs EPROM & EEPROM

Intel's ETOX™ II (EPROM Tunnel OXide) flash memory is a single-transistor cell providing nonvolatile storage like EPROM, with electrical erase similar to EEPROM. Reprogramming flash memory entails electrically erasing all data bits in parallel, then randomly programming data into any byte in the array. The programming operation is achieved via channel hot electron injection (CHE), just like EPROMs. Flash electrical erasure, however, is accomplished through Fowler-Nordheim (FN) tunneling. Using separate program and erase methods (CHE vs. FN Tunneling), in different cell locations, drain vs. source, permits process optimization for high cycling endurance—the number of complete erase and re-writes. Traditional low-density EEPROMs tunnel through the same memory cell junction for both programming and erasure. Because EEPROMs erase before programming each byte, these processes must occur very fast. Therefore, internal voltages used to program or erase 5V-only EEPROM memory cells are high (e.g., 18V–30V). The combination of higher voltage with programming and erasing through the same junction contributes to EEPROM's oxide breakdown, poor data retention and reduced cycling capability.

Intel's flash memory erasure (tunneling) voltage is below the critical oxide breakdown voltage. By using block erasure instead of EEPROM's byte erasure, erase times are relaxed, reducing tunneling voltages. Programming Intel's Flash Memory is non-destructive to the floating gate oxide compared to EEPROM's use of tunneling for programming. These features for erase and programming provide Intel's Flash Memory with the highest endurance (typically over 100K cycles) compared to that of traditional EEPROM cycling. Furthermore, flash memory exhibits lower failure rates at any given cycle count.

2.2 Program and Erase Automation

The 28F001BX integrates a Write State Machine (WSM) on-chip to internally implement program and erase algorithms. Operations are initiated through command sequence writes to the Command Register, and progress is reported back to the user through Status Register bits. Software timers are no longer required, as timing is now regulated through the on-chip oscillator. System software requirements are decreased in comparison to past algorithms, minimizing overhead and development effort, and allowing code execution and interrupt servicing while simultaneously programming or erasing the device.

The Erase Suspend command halts block erase to execute code or read data from any other block. This feature gives the system the capability to service higher level interrupts requiring data from the 28F001BX during the erase interval. After issuing the Erase Resume command, the WSM continues erase where it left off when suspended.

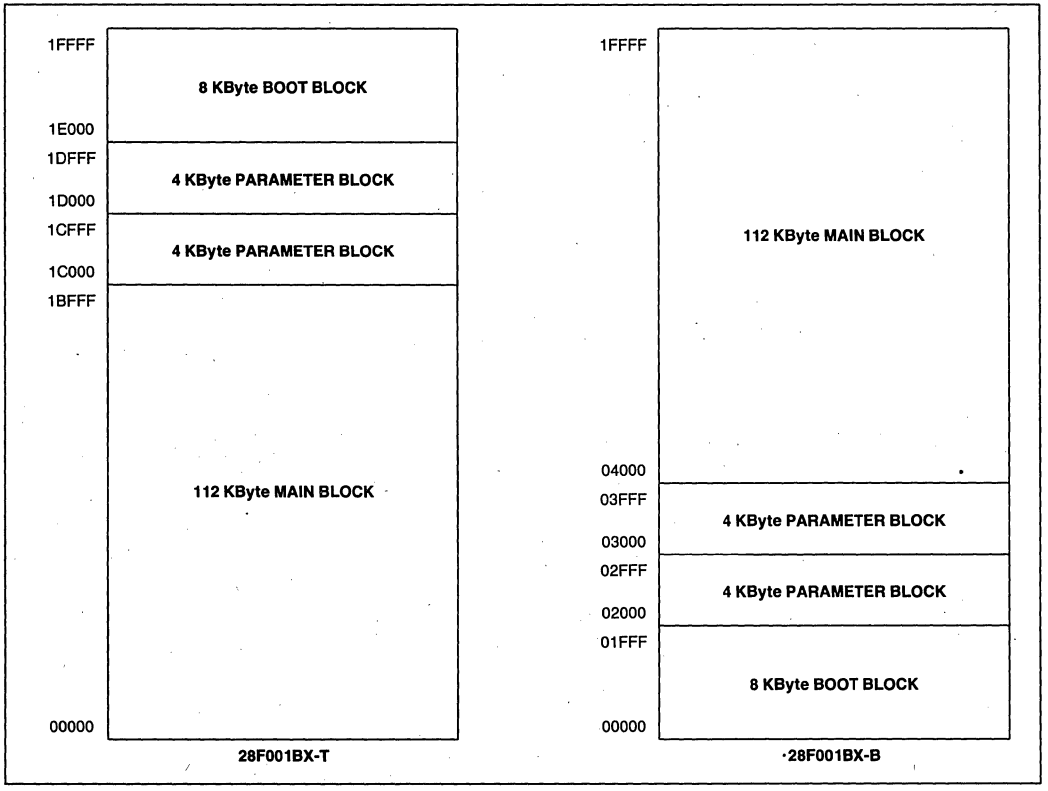


Figure 1. 28F001BX Memory Maps

2.3 Blocked Architecture

The 28F001BX family combines the safety of a hardware-protected 8 KByte "boot" block with the flexibility of two 4 KByte "parameter" blocks and one 112 KByte main block. Each block can be individually erased and programmed without affecting code stored in another block, ensuring data integrity. The boot block is intended to contain secure code which minimally will bring up the system and download code to the other blocks of the 28F001BX if required. Once programmed, it is hardware-locked from further alteration, guaranteeing true non-volatility.

The 28F001BX-T's lockable block location provides compatibility with microprocessors and microcontrollers that boot from the top of the memory map, such as Intel's x86 family and i860™ microprocessor. The segmentation of the 28F001BX-B is identical. Its lockable block location provides compatibility with microprocessors that boot from low memory, such as Motorola and AMD products. See Figure 1 for illustrations of the two memory maps available in the 28F001BX family.

The two 4 KByte parameter blocks have multiple uses in BIOS environments. They can be used to back up the CMOS setup parameters such as floppy and hard disk type, processor speed, system memory size, graphic display type and presence of a coprocessor. Today, should the system battery fail, the user loses information in battery-backed SRAM. The non-volatile parameter blocks provide the system capability to automatically back up and recover this information. Also, EISA systems can store software variable information such as add-in board addresses, DMA channels and interrupt values/levels.

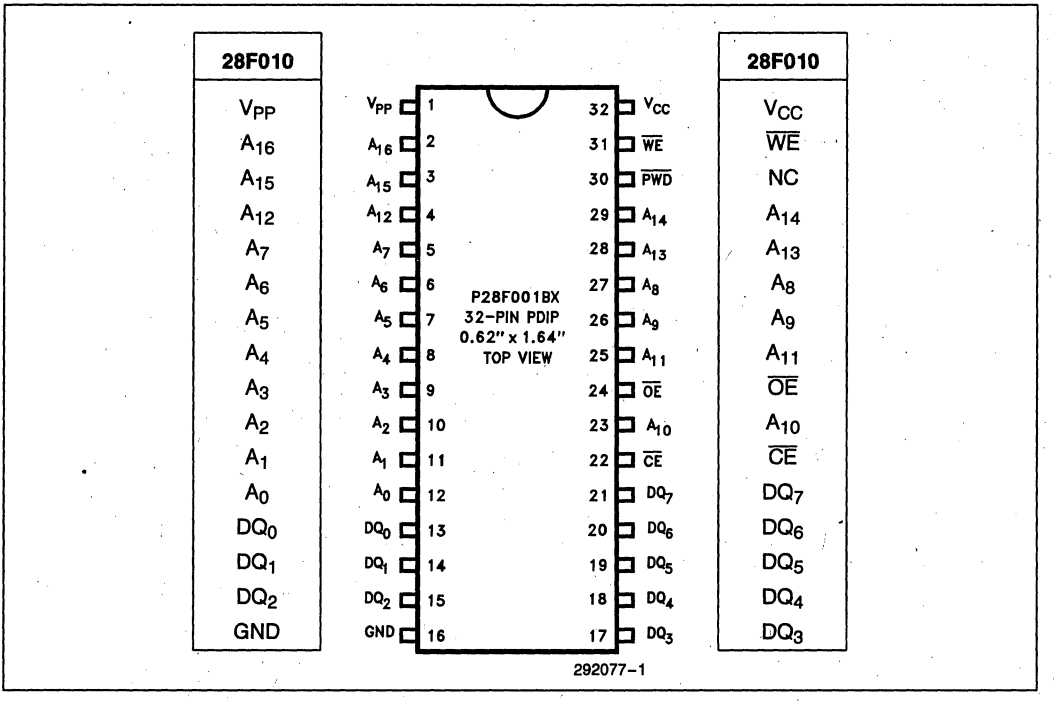


Figure 2. 28F001BX DIP Pin Configuration

2.4 Deep Powerdown Mode

Market analysts predict that the high-growth segments of the PC industry for the next several years will be in the laptop, palmtop and handheld product lines. Power management software is becoming an integral part of PC system BIOS as manufacturers attempt to squeeze the maximum amount of system operation time from their battery pack power supplies. A key indicator of this trend is Intel's i386™SL microprocessor superset, which adds hardware power management to the Intel386™ architecture and answers the needs of low-power applications.

The 28F001BX family features deep powerdown capability and is ideally suited for these same battery-operated systems. Powerdown is entered through low voltage on the $\overline{PWD}$ pin. Typical power consumption through V_{CC} in powerdown is 0.25 μW, regardless of power supply voltages and activity on the external bus. Once BIOS is shadowed to system DRAM for high-speed execution, the 28F001BX can be shut down for minimal current drain.

This same pin, with 12V present on it, gates program and erase of the boot block. Such a hardware lockout preserves the system boot code once initially programmed and protects it from inadvertent alteration or computer virus compromise.

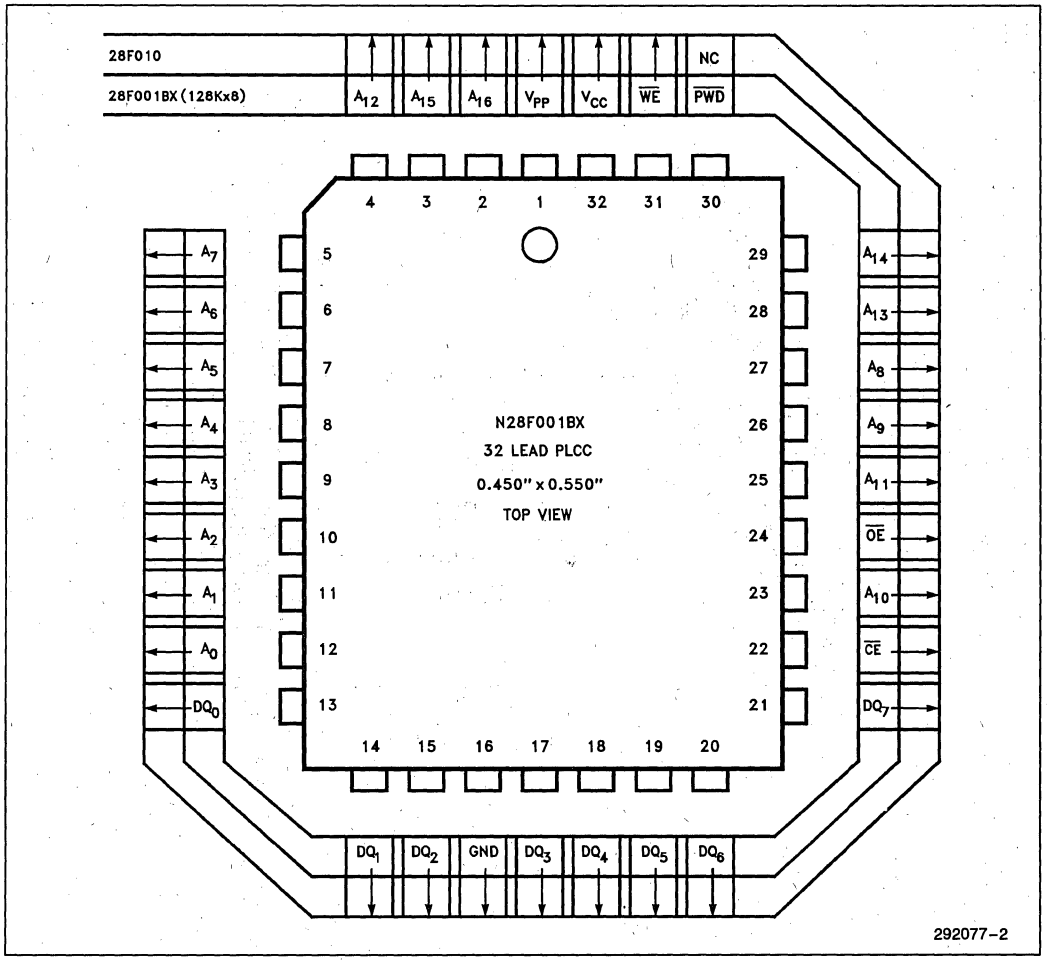


Figure 3. 28F001BX PLCC Lead Configuration

2.5 28F001BX Pinouts, Physical Layout and Upgrade

Intel's 28F001BX is offered in three standard 32-pin packages: Plastic Dual In-line Package (PDIP), Plastic Leaded Chip Carrier (PLCC), and Thin Small Outline Package (TSOP). All three pinouts provide backward compatibility with Intel's 28F010 bulk-erase flash. See Figures 2, 3, and 4 for pinout details.

Plastic Dual In-Line Package

PDIPs with sockets provide an excellent way to prototype and debug new designs. The 28F001BX is backward pin-compatible with 1 Mbit standard flash and EPROMs.

Plastic Leaded Chip Carrier

Most system designs today require surface mount technology (SMT) due to shrinking board real estate and portable form factors. PLCC is one SMT component that uses as little as 35% of the overall board space compared to PDIP. Its small size is attributed to the center-to-center lead spacing of 50 mils versus 100 mils, as well as its four-sided pinout. The J-lead design allows the PLCC to be directly soldered to the circuit board. Most SMT manufacturing equipment can easily handle the PLCC's 50-mil lead pitch. PLCC SMT sockets such as that offered by AMP (P/N 821977-1) have an identical foot-print for 32-pin devices. Such sockets can be used in place of directly soldering a PLCC for prototype build and code testing. Once the reprogramming code is tested and debugged, flash PLCCs can then be surface-mounted without socketing during production runs.

Thin Small Outline Package

TSOP is the package of choice for hand-held equipment or palmtop/laptop computers. These compact systems require minimal height and area for all components, for which TSOP excels. TSOP height measures 1.2mm versus 3.5mm for PLCC. TSOP area is 8mm x 20mm compared to PLCC's 11.43mm x 13.97mm. Therefore, TSOP has significantly less total volume: TSOP = 172.8mm^3, while PLCC = 656.3mm^3, and DIP = 1872.3mm^3. State-of-the-art center-to-center terminal spacing of 20 mils yields a smaller package with narrower conductor traces than PLCC or PDIP. Location of pins on both ends of the package allows traces for TSOP to be routed underneath the chip, reducing board layers. TSOP for the 28F001BX is available in the standard (E) pin configuration. For multiple chip flash systems, Intel's bulk-erase 28F010 flash memory is available in both standard (E) and reverse (F) pin configurations (see Figure 5) allowing components to be laid out end-to-end and side-to-side for highest board density (see Figure 6). Note how pins 32–17 on the standard pinout match pins 1–16 on the reverse pinout, and how pins 1–16 on the standard pinout match pins 32–17 on the reverse pinouts.

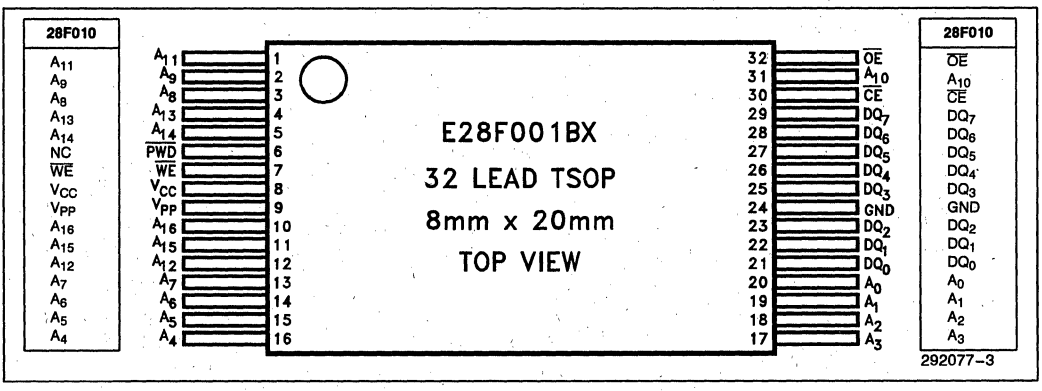


Figure 4. 28F001BX TSOP Lead Configuration

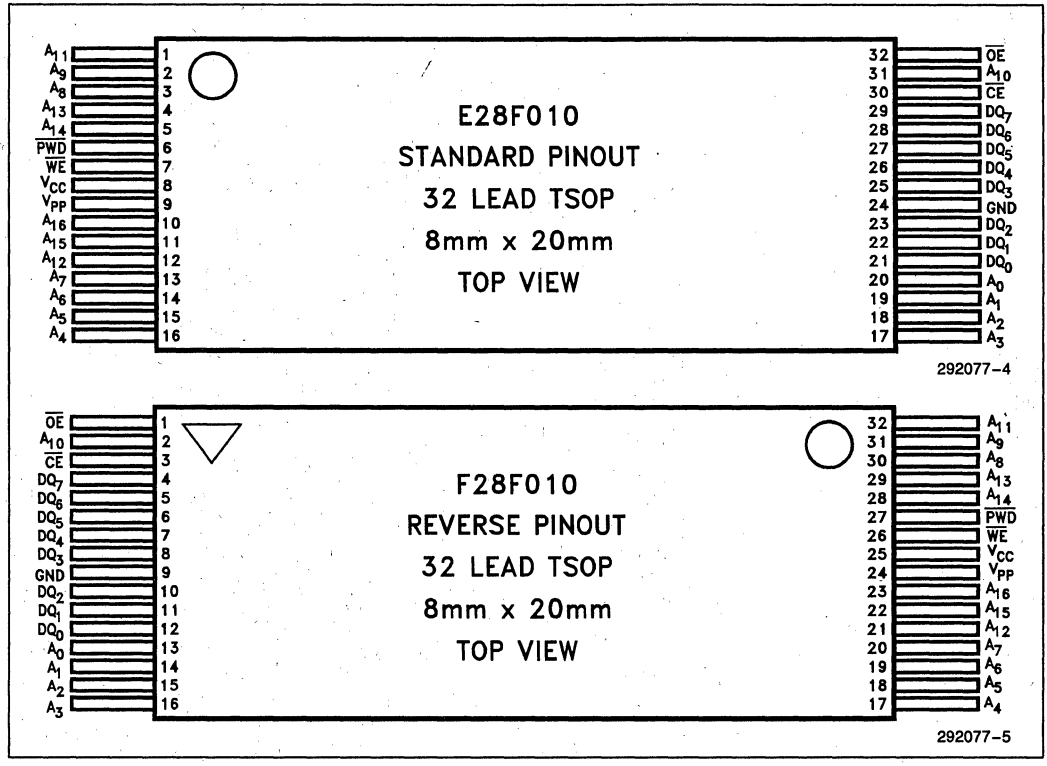


Figure 5. 28F010 Standard and Reverse TSOP Lead Configurations

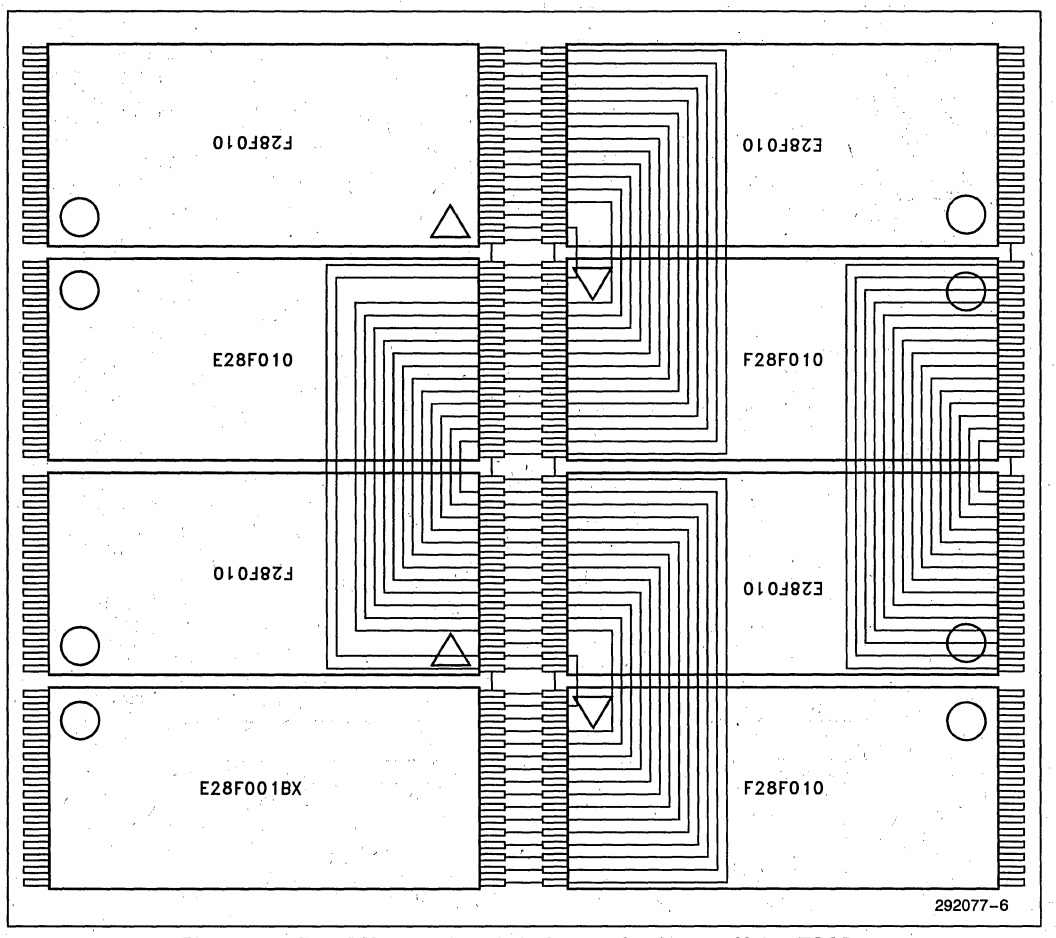


Figure 6. 28F001BX and 28F010's in Serpentine Layout Using TSOPs

2.6 V_{PP} Specifications

Fixed V_{PP} and V_{CC}

Flash memories, like EPROMs, require a 12V programming power supply. Unlike EPROMs, however, the V_{PP} for flash memory is a fixed, standard level. When combined with the Command Register erase/program control, Intel flash memories use a simple, SRAM-like hardware interface with standard microprocessor timings.

Intel's Flash Memory V_{PP} specification is 12.0V ±0.6V (5%), compatible with most off-the-shelf system power supplies. The IBM PC technical reference manual specifies the 12V power supply at 12.0V, +5% and −4%. Additionally, some hard and floppy drives require 12V ±5%. Therefore, most PC power supplies have 12V supplies with ±5% or better tolerance. Possible exceptions to this are laptop and/or palmtop PCs. Some of these require 5V-only designs, in which case 5V is charge-pumped to 12V. It is essential to use the specified V_{PP} when programming and erasing flash. Once the commands to program or erase are issued, the device internally derives the required voltage references from the V_{PP} supply. Therefore, an improper V_{PP} level degrades the performance of the part.

The Write State Machine automatically monitors the voltage present on the V_{PP} pin, beginning when program or erase setup commands are issued and continuing throughout the internal algorithm interval. If low V_{PP} is detected, the WSM automatically aborts the program or erase attempt and reports an improper voltage error to the user through the Status Register. The hardware design section discusses various methods of V_{PP} generator if your 12V power supply does not meet the proper tolerances or 12V is not available.

$\overline{PWD}$, V_{CC} and V_{PP} Lockout Protection

The deep powerdown $\overline{PWD}$ pin provides hardware write protection for 28F001BX flash memories. Until this pin transitions to TTL-level V_{IH}, write attempts to the device Command Register are ignored, regardless of power supply levels or activity on the system bus and control inputs. Typically, the system designer will gate this signal with a system POWER GOOD indicator output to ensure system stability before memory accesses begin.

The 28F001BX family provides additional protection for designs that tie 12V directly to the device. Since the 12V supply is less capacitively loaded than the 5V supply, the 12V power supply reaches full value faster during power-on. If Command Register lockout protection was not provided, a finite possibility exists that inadvertent writes may occur during power-on. For this case, Intel's 28F001BX flash memory supplies Command Register lockout protection when V_{CC} is below 2.5V, preventing writes to flash memory from occurring. Since CMOS logic is valid at 2.0V, a 0.5V margin of protection exists, providing extra time for control signals to settle before the Command Register is activated. Program/erase inhibit is guaranteed with V_{PP} below 6.5V, with corresponding V_{PP} low reported through the Status Register. Once V_{CC} reaches 2.5V, the Command Register begins processing valid commands, and program/erase attempts may initiate with V_{PP} greater than 6.5V. At this point, the system is responsible for write control.

When the 28F001BX V_{CC} powers up, or after the $\overline{PWD}$ pin transitions to V_{IL} and back to V_{IH}, the Command Register is automatically initialized to Read Array mode.

3.0 HARDWARE DESIGN CONSIDERATIONS

The system level hardware requirements for implementing BIOS and application storage in flash are:

- Write Enable available to all of the flash memory
- 12V routed to flash location or generated on-board
- CMOS control-signal interface, or $\overline{PWD}$ gated by a power-good signal
- Data buffer or transceiver that works in both write and read directions
- Space in memory map allocated for each application's size

Intel's i386SL microprocessor superset was chosen for the design example, shown in Figure 7. The Intel386SL microprocessor superset integrates all major components of a Intel386 based design on two chips, including bus memory, cache controllers and the ISA peripheral subsystem. Additionally, it consolidates hardware power management for battery-operated designs such as laptop, handheld and palmtop computers.

Note the clean interface between the superset and 28F001BX-T. Flash memory was comprehended early in the design of the Intel386SL microprocessor superset, to ensure a minimal-chip interface. Transceivers for the system bus, as well as a flash memory $\overline{CE}$ signal, are integrated on the Intel386SL.

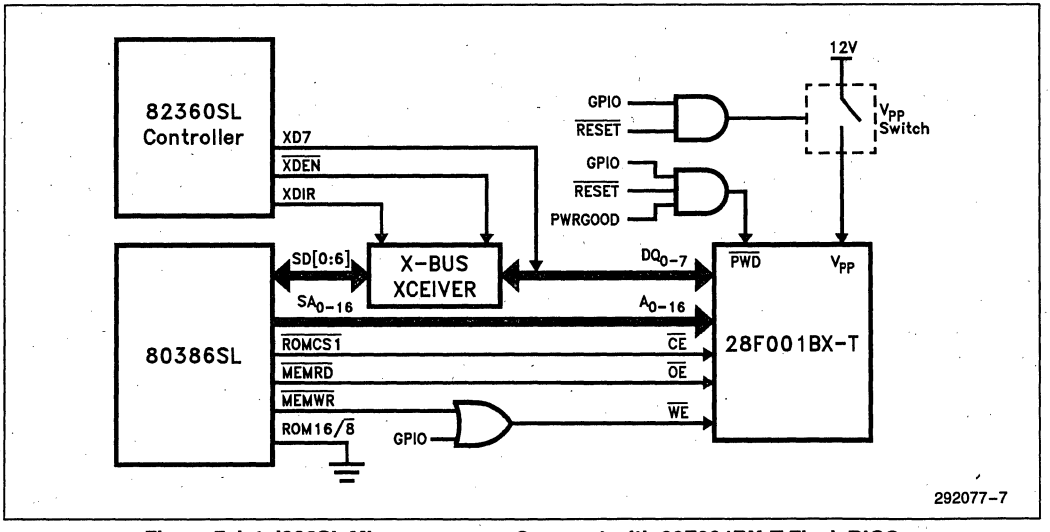


Figure 7. Intel386SL Microprocessor Superset with 28F001BX-T Flash BIOS

The $\overline{PWD}$ pin is gated by a power good signal to ensure control logic integrity before writes to the 28F001BX-T are allowed. It is also gated by System Reset, to abort program or erase if required, and by a separate General Purpose Input-Output (GPIO) line to power down the device once BIOS is shadowed to RAM. CMOS logic will guarantee lowest power dissipation.

Similarly, system 12V is gated both by System Reset and a GPIO line. Software can switch 12V to the 28F001BX-T only when programming or erasing it, minimizing system power consumption. V_{PP} generation and switching methods are discussed in Section 3.3.

Application code, assuming a ROM in the BIOS socket, is sometimes designed to write to BIOS locations to generate software timing delays (versus using NOPs). Gating $\overline{WR}$ to the flash memory with a GPIO line disables writes until desired by BIOS update software.

3.1 BIOS Boot Code Requirements and System Configurations

The previous design assumed that shadow RAM was available in the system. Referencing Figure 8, we see that the BIOS is actually stored in the main block of the 28F001BX, from system address E0000H–FBFFFH. In this scenario, the processor jump vectors, BIOS checksum and recovery code are stored in the 8 KByte boot block. This is the area the processor will jump to on powerup or after reset. The boot block code will execute a checksum check of the main block for a valid BIOS. If successful, the processor will check system RAM, copy the main block code to high memory DRAM and jump to this area for the remainder of Power On Self Test (POST), as well as further BIOS calls. Optionally, the 28F001BX can then be disabled for power savings.

If BIOS checksum determines an invalid BIOS, the system RAM and floppy drive (or possibly modem) are initialized using the boot block recovery code. The system requests (through screen display or speaker "beeps") that the user install the BIOS update floppy disk. A search of the floppy disk is made for a specific file name, and once found, update code is used to re-initialize the main BIOS block. System reboot restores normal operation. Alternatively, the BIOS recovery code can contain specific, non-DOS sector/track information pertaining to the location of the new BIOS update file. Thus, the file is protected and not readable to basic DOS users.

If ROM BIOS disable is overriden by system software or the user (through setup utility), the design must compensate for the altered BIOS location to prevent BIOS calls jumping to incorrect code locations. The following two methods provide alternative solutions for the system designer.

Address Shift Configuration

In this scenario, after BIOS initialization is complete, a write to a latch, register or flip-flop shifts addresses for future BIOS code fetches by 16 KBytes. This allows the system to correctly access the main block and bypass parameter and boot blocks. A system reset or loss of power clears this latch, allowing booting from the boot block once again. Figure 9 shows the input and output signals required for a µPLD address shifter. It shifts addresses in the range FFFFFH–E4000H (112 KBytes) to FBFFFH–E0000H.

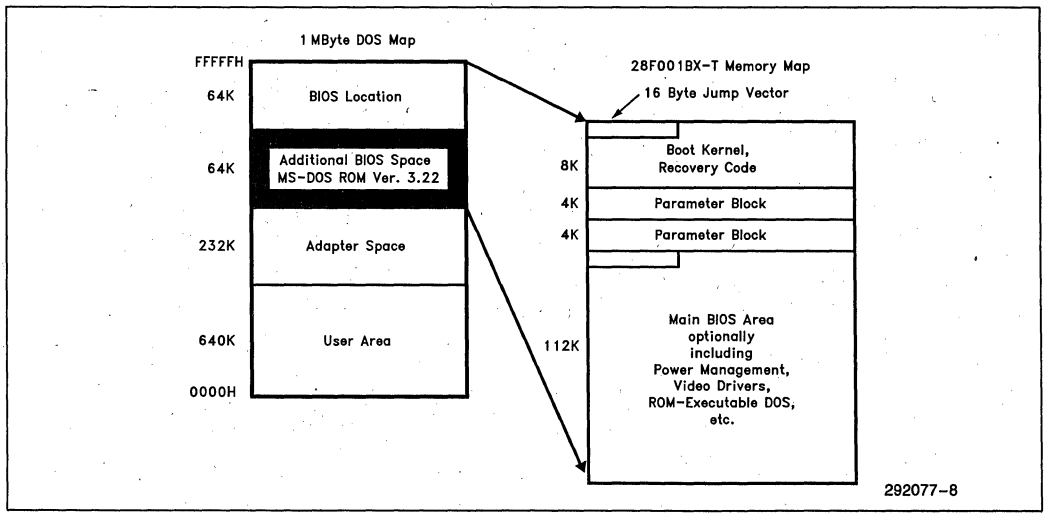


Figure 8. 28F001BX-T in 1 MByte DOS Memory Map

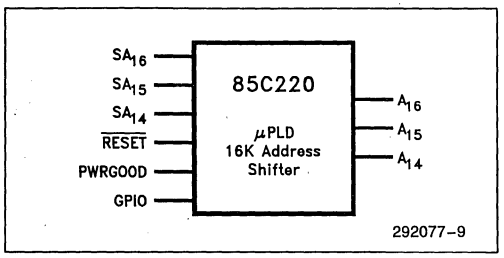


Figure 9. Address Shift Circuitry

Address Inversion Configuration

Figure 10 presents an alternative approach to configuring the 28F001BX in the system memory map. Simple inversion of address line A_{16} to the 28F001BX moves the boot block to the lower half of flash memory as seen by the system. In normal operation, the processor boots and executes from the main array areas, which store the system BIOS, video BIOS and/or DOS in ROM.

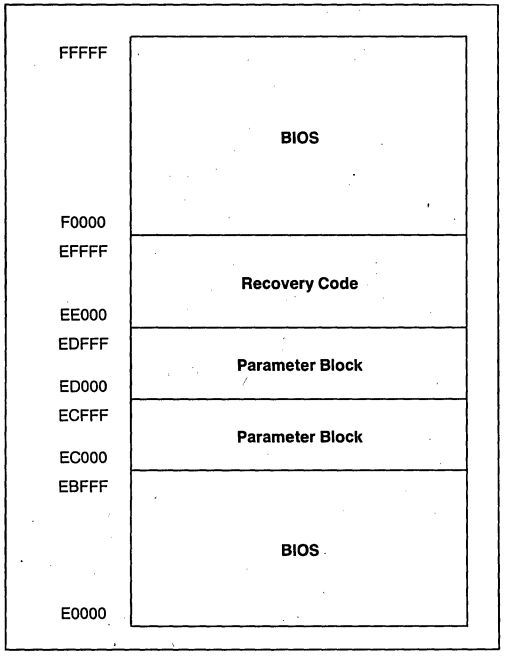


Figure 10. Inversion Configuration (Normal)

If power loss aborts a BIOS update, the main array block will be partially programmed/erased and the code in this block unusable. The system will "hang" or not boot at all. To boot from the boot recovery block, restore address A_{16} polarity, producing the memory map shown in Figure 11. A keyboard sequence, switch on the back of the PC or jumper on the motherboard can toggle A_{16} restore logic and "un-invert" it. After reconfiguration, the processor boots from the boot block and executes its recovery algorithm to restore main array block contents. Re-inverting A_{16} reinstates normal system bootup and operation.

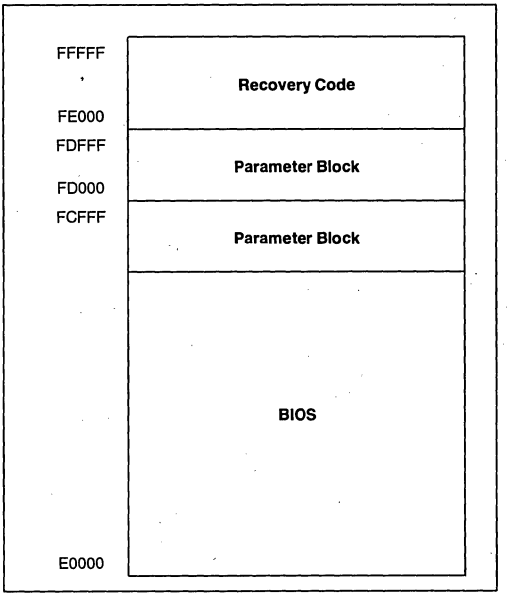


Figure 11. Inversion Configuration (Recovery)

Since standard BIOS code does not support boot block recovery, your BIOS software engineers must design the recovery code for the 8 KByte block. See Section 4.6 for a flowchart of an example recovery algorithm. Third-party BIOS vendors, working with Intel, have also developed recovery code for the 28F001BX (see Appendix C). With the exception of this recovery section, the rest of the BIOS remains the same.

3

3.2 V_{PP} Generation

For flash BIOS designs, the 12V V_{PP} can be provided by:

1. Using the existing 12V supply from PC Power Supply, or
2. Generating 12V using a charge pump or DC–DC converter from the 5V supply.

Flash typically requires only 10 mA for program or erase (30 mA max); otherwise only 10 μA is drawn in standby mode, and 0.8 μA in deep powerdown mode.

Using System 12V Directly

As stated earlier, the IBM PC technical reference manual specifies the 12V supply as +5% and −4%, which meets the Intel Flash Memory V_{PP} requirement. If your power supply meets this condition and has CMOS logic, 12V from the PC power supply can be tied directly to flash memory, eliminating the need to add extra circuitry for V_{PP} generation. This is possible due to the $\overline{PWD}$, V_{CC} and V_{PP} write lockout protection offered in the 28F001BX.

Pumping 5V to 12V

If your system does not provide 12V or does not meet flash memory specifications, several 5-to-12V converters are available, including surface-mount versions. Application Note AP-316, available from your local Intel Sales Office, lists several V_{PP} solutions which offer on/off control of V_{PP} and provide a steady V_{PP} rise and little overshoot. Figure 12 shows one example. On power-up, system reset or when V_{CC} is below 4.5V, V_{PP} is forced off. It is enabled (or disabled) by writing to the $\overline{V_{PP}EN}$ I/O port address. On/off capability is essential for battery-operated equipment and eliminates the need for $\overline{WE}$ filtering. The $\overline{V_{PP}EN}$ signal "OR'ed" with the system memory write ($\overline{MEMWR}$) functions as the clock signal for the 74FC74 D-flip-flop. The D-input is latched when $\overline{MEMWR}$ goes high. Writing a one or a zero turns V_{PP} on or off, respectively.

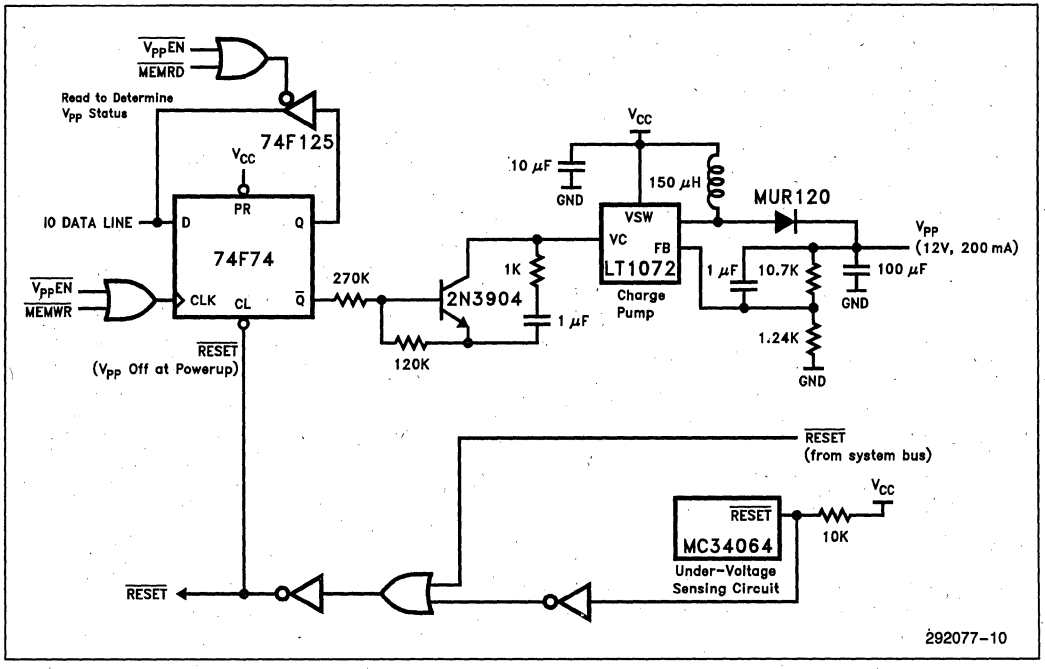


Figure 12. V_{PP} Generation with Write Protection

Linear Technology's LT1072, a switching regulator, is used as a 5V to 12V charge pump. The 10.7K and 1.24K resistors are used to establish the correct reference voltage to obtain 12V. The 100 μF capacitor at the output can handle up to 200 mA. For a single- or double-chip BIOS design, this capacitor value can be halved or even quartered to allow selection of a SMT capacitor value, since the maximum I_{PP} current per device is only 30 mA (10 mA typical). Allow sufficient time when switching V_{PP} on, letting the charge pump level out and enabling the Command Register to receive program or erase commands. The diode, MUR120, keeps the inductor from absorbing current from the charged output capacitor.

Security

Controlling V_{PP} provides the benefit of system hardware security. Beyond this, you can design for even higher security levels. The first level could be the design of a simple software password routine that would only turn on V_{PP} when a correct password is given. Alternatively, you can provide a jumper to allow 12V to the part for a BIOS update and then return it when reprogramming is finished. The system should check this pin to see if the jumper was left in the programming position and remind the user to move it. Unless V_{PP} is at 12V, the flash memory contents cannot be changed and acts just like ROM. Disabling V_{PP} until voltages have stabilized provides additional power-up protection.

The Motorola component, MC34064, is an under-voltage sensing circuit that begins functioning when V_{CC} is above 1V. Between 1V and 4.6V, the $\overline{\text{RESET}}$ output is active. This (or a system $\overline{\text{RESET}}$) clears the 74FC74, keeping V_{PP} off. Alternatively, if you use CMOS logic, you could make use of Intel's flash memory V_{CC} and V_{PP} lockout functions. While V_{CC} is below 2.5V, the Command Register is locked out. Since CMOS control logic is active at 2.0V, a 0.5V safety margin exists for control logic to settle down before the part becomes active. Program and erase attempts are inhibited with V_{PP} below 6.5V. For both CMOS and non-CMOS designs (i.e., control logic active at 2.0V), gate $\overline{\text{PWD}}$ with the power supply's "Power Good" signal or the MC34064's $\overline{\text{RESET}}$ output (Figure 13). Until $\overline{\text{PWD}}$ transitions to V_{IH}, the part ignores all write attempts, regardless of power supply voltages and bus activity.

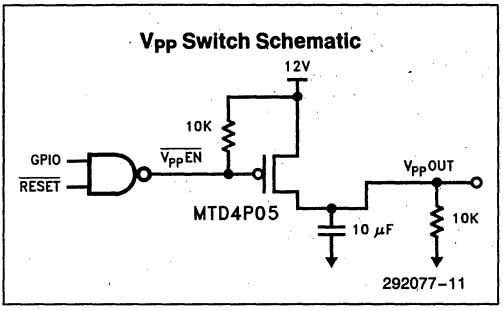


Figure 13. V_{PP} Switch Using MTD4P05

Using a MOSFET Switch

For laptops or palmtops, an always-active 12V may not provide acceptable power management. For these systems, a MOSFET switch will toggle 12V to the flash memory, minimizing current draw when not needed. Several DC switches exist, but there are a few issues to consider in your selection. Choose a switch with low "ON" resistance to keep the V_{PP} voltage within flash memory tolerances. The system 12V power supply must be specified to a tighter range to allow for any voltage drop through the switch. Allocate an I/O line (V_{PP} enable) to turn the switch on and off. To handle "warm RESETS", the V_{PP} enable must be gated with the system $\overline{\text{RESET}}$ line. The Motorola MTD4P05 is one example of a surface-mount switch with low drain-source resistance. Assuming a 12V + 5% and −4% supply:

R_{DS} = 0.6Ω

I_{PP} = 30 mA (Worst Case)

ΔV_{SWITCH} Drop = (30 mA x 0.6Ω) = 0.02V, ≪ (4% of V_{PP} =) 0.48V.

Figure 13 shows a schematic of a V_{PP} switch design.

3.3 Modifying an Existing Motherboard

EPROM/ROM Designs

If you are modifying an existing motherboard design for a flash memory BIOS, there are a few things you should consider. First, check the logic design to determine if $\overline{\text{WR}}$ is decoded and connected to the BIOS EPROM location. Typical motherboard logic designs do not allow writes to the EPROM locations and treat EPROM writes as invalid (e.g., $\overline{\text{ROMCS}}$ not generated with $\overline{\text{MEMWR}}$). This is overcome by generating the BIOS location's $\overline{\text{WR}}$ externally by either adding the necessary discrete logic or adding a 3-to-8 decoder (see Figure 14 for an example). In either case, tap into the M/$\overline{\text{IO}}$ and $\overline{\text{WR}}$ control lines and configure the decoder to provide a logic low for the M "AND" $\overline{\text{WR}}$ "AND" BIOS address condition.

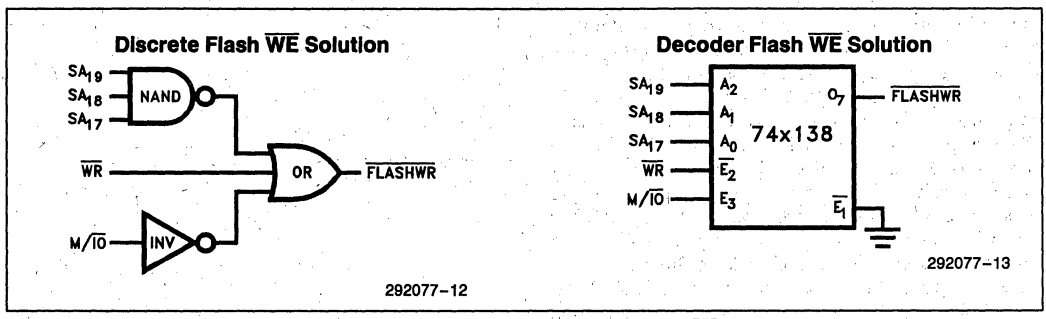


Figure 14. Discrete and Single-Chip Decoder WE Solutions

Secondly, check to see if the BIOS code transceiver or buffer for the EPROM location works in both directions. The transceiver may need a special BIOS call to unlock it in the "write" direction, or you may have to reprogram the logic for that portion of your board. If your chip set data buffer works only in one direction, a transceiver and direction logic must be added to the CPU bus to pass data to and from flash memory.

Your system must also be capable of routing 12V to the BIOS socket for program and erase. Optionally, provide capability for deep powerdown mode through $\overline{PWD}$ (see Section 3.0), or simply tie $\overline{PWD}$ to V_{CC} via a jumper, blue wire or trace modification to the motherboard. Finally, address inversion or shift mechanisms outline in Section 3.1 can optionally be added for recovery capability with the 28F001BX.

28F010 Flash Memory Designs

If your design currently incorporates Intel's 28F010 flash memory, hardware upgrade to the 28F001BX is simple. Transceiver, BIOS write and V_{PP} requirements will have already been considered in the original design. Gate the $\overline{PWD}$ input if powerdown capability is desired, or jumper this pin to V_{CC}. Finally, invert or shift the system addresses as in Section 3.1 if BIOS "ROM" access after shadowing to DRAM is anticipated.

3.4 In-System Write vs On-Board Programming

When devices are soldered directly to a printed circuit board, one of two sources control flash memory reprogramming:

1. the system's own processor, or
2. a PROM programmer connected to the board.

These options are called In-System Write (ISW) and On-Board Programming (OBP), respectively. Their respective benefits are discussed in detail in AP-316.

With ISW, the system drives the reprogramming process and generates V_{PP} locally. Under this scenario, the board manufacturer will initially program at least the boot block in a PROM programmer. This removes the need for circuitry on-board to unlock the boot block, guaranteeing boot code integrity throughout system life. A good design practice for ISW-type designs is to socket the first few flash BIOS prototypes. SMT-only designs can also socket using PLCC SMT sockets. Socketing enables the system designer to easily work out any bugs with in-system flash reprogramming by allowing the removal of a flash part for external reprogramming in a PROM programmer. Once ISW reprogramming is fully debugged, pre-programmed flash parts can be soldered directly to the circuit board without a socket. All flash memory components are exposed to a data-retention bake testing and checked for any data loss before shipping. It is extremely unlikely that data in a production flash device can be corrupted from heat by a production-run soldering application.

OBP uses an external board programmer to supply V_{PP} and V_{HH} and control the programming process. Certain design considerations must be evaluated prior to laying out the design. Some manufacturers using TSOP may also want to remove a handling step from the manufacturing process by providing the capability to program flash for the first time after being soldered directly onto the circuit board. OBP can accomplish this if the design is first laid out correctly to support OBP. External circuitry generates voltages needed to unlock and program/erase the boot block.

3.5 Ideas for Using Extra Adaptor Space

Laptop and palmtop systems may have adaptor space available in the system memory map since there typically isn't much room for add-in boards. Additionally, they may not use up the entire 128K of BIOS space due to their fixed feature set and limited upgrade capability. This extra memory space can hold ROM executable programs like Lotus 123, WordPerfect, Microsoft Works, etc. Using Intel's flash TSOPs, a small application cache can reduce a laptop's disk access and increase battery life.

Additionally, ROM-Executable DOS can be placed anywhere in adapter space. For example, MS-DOS ROM Version 3.22 requires 62 KB of adaptor space today (this may change on subsequent revisions). One location for MS-DOS ROM Version 3.22 is directly under the BIOS (again see Figure 8). Today's typical BIOS consumes 64 KB or less; consequently, both the BIOS and MS-DOS ROM Version 3.22 could reside in a single 28F001BX (128 KBytes), yielding reduced chipcount. However, if power management code is added to the BIOS, system BIOS code could grow to 80 KB or more. Therefore, designs that include both power management and MS-DOS ROM Version 3.22 should consider using both a 28F001BX and 28F010 flash device (or two 28F001BX's). This leaves extra space for BIOS and MS-DOS ROM to grow in the design, while providing additional storage for the video BIOS.

4.0 SOFTWARE DESIGN CONSIDERATIONS

Intel's Flash Memory provides a cost-effective, updatable, nonvolatile code storage medium. The 28F001BX integrates the Quick-Pulse Programming and Quick-Erase algorithms of prior Intel Flash Memories on-chip, using the Command Register, Status Register and Write State Machine (WSM). On-chip integration dramatically reduces system overhead, simplifies system software creation and debug and provides SRAM-like timings to the Command and Status Registers. WSM operation, internal program/erase verify and V_{PP} high voltage presence are monitored and reported via appropriate Status Register bits. Table 1 lists the 28F001BX command set, while Table 2 details the Status Register bits and their meanings.

Table 1. 28F001BX Command Definitions

Command	Bus Cycles Req'd	Notes	First Bus Cycle			Second Bus Cycle		
		1, 2	Operation	Address	Data	Operation	Address	Data
Read Array/Reset	1		Write	X	FFH			
Intelligent Identifier	3	1, 2, 3	Write	X	90H	Read	IA	IID
Read Status Register	2	2	Write	X	70H	Read	X	SRD
Clear Status Register	1		Write	X	50H			
Erase Setup/Erase Confirm	2	1	Write	BA	20H	Write	BA	D0H
Erase Suspend/Erase Resume	2		Write	X	B0H	Write	X	D0H
Program Setup/Program	2	1, 2	Write	PA	40H	Write	PA	PD

NOTES:
1. IA = Identifier Address; 00H for manufacturer code, 01H for device code.
   BA = Address within the block being erased.
   PA = Address of memory location to be programmed.
2. SRD = Data read from Status Register. See Table 2 for a description of Status Register bits.
   PD = Data to be programmed at location PA. Data is latched on the rising edge of $\overline{WE}$.
   IID = Data read from intelligent identifier.
3. Following the intelligent identifier command, two read operations access the manufacturer and device codes.
4. Commands other than shown above are reserved by Intel for future device implementations and should not be used.

Table 2. 28F001BX Status Register Definitions

WSMS	ESS	ES	PS	VPPS	R	R	R
7	6	5	4	3	2	1	0

SR.7 = WRITE STATE MACHINE STATUS
1 = Ready
0 = Busy

SR.6 = ERASE SUSPEND STATUS
1 = Erase Suspended
0 = Erase In Progress/Completed

SR.5 = ERASE STATUS
1 = Error In Block Erase
0 = Successful Block Erase

SR.4 = PROGRAM STATUS
1 = Error In Byte Program
0 = Successful Byte Program

SR.3 = V_{PP} STATUS
1 = V_{PP} Low Detect; Operation Abort
0 = V_{PP} OK

NOTES:

The Write State Machines Status Bit must first be checked to determine program or erase completion, before the Program or Erase Status bits are checked for success.

If the Program AND Erase Status bits are set to 1's during an erase attempt, an improper command sequence was entered. Attempt the operation again.

If V_{PP} low status is detected, the Status Register must be cleared before another program or erase operation is attempted.

The V_{PP} Status bit, unlike an A/D converter, does not provide continuous indication of V_{PP} level. The WSM interrogates the V_{PP} level only after the program or erase command sequences have been entered and informs the system if V_{PP} has not been switched on. The V_{PP} Status bit is not guaranteed to report accurate feedback between V_{PPL} and V_{PPH}.

SR.2–SR.0 = RESERVED FOR FUTURE ENHANCEMENTS
These bits are reserved for future use and should be masked out when polling the Status Register.

The WSM on-chip oscillator internally times the program/erase algorithms, making software timers unnecessary. Block precondition is also controlled by the WSM as part of the erase algorithm. Block data programming to "0's" before erasing is no longer needed.

Intel's high quality design, manufacturing and testing result in outstanding reliability and performance throughout device life. Although Program Status and Erase Status bits are provided for Status Register completeness, errors will probably not be encountered, if proper V_{PP} levels and software sequences are implemented.

Intel offers standard software drivers, written in "C", to assist software engineers implementing 28F001BX reprogramming for update utilities. These high-level routines, found in Appendix A, are adaptable to a wide range of μP and μC platforms and system architectures.

Covered in this section are the major software steps for a flash BIOS update utility:

- Update software for a modified system
- Pseudo-Code overview
- Initializing the system
- Code loader routine
- Flash re-programming
- Recovery routines
- Power management

4.1 Update Software for a Modified System

The design example of Section 3.0 assumes BIOS shadowing for BIOS code execution while allowing BIOS writes to the flash socket. Many systems provide a register which enables BIOS writes and reads. Some systems may not allow BIOS reads from RAM while performing BIOS writes to the flash socket, or vice versa. The reasons may be simple; no shadow RAM exists in the system (8088 or 8086 systems), or system logic treats "ROM writes" as an invalid operation. In these cases, perform all your required BIOS calls before you erase and program the flash memory. But keep in mind, to update the user on the progress of flash programming and indicate when programming is finished, you should add some basic screen or speaker "beep" routines to your update utility.

4.2 Pseudo-Code Overview

The following pseudo-code for an update utility provides a brief description of the process of updating a BIOS in-situ. It is based on software developed by a customer for a PC platform with BIOS update capability. This Intel386-33 MHz system uses the 28F001BX for BIOS storage. Modify the flowchart below, if needed, for your particular chipset and hardware environment.

Pseudo-Code for Flash Update Routine

```
Initialize system (set up user screen, check battery power, check device ID)

Get BIOS file options (from floppy or modem)
    If no file present
        Send error message to insert BIOS update floppy, or press ESC to exit

Display BIOS update files, prompt user for choice and load to memory
    If file invalid,
        Prompt for file or exit

Inform user what is about to happen, with option to continue or exit
    If user continues, inform them to not turn off the power or soft-reboot (CNTL-ALT-DEL)

Erase 28F001BX main/parameter blocks
    If system interrupt occurs
        Suspend erase if flash memory access is required
            Resume Erase

Write file[s] into flash memory
Indicate to user that flash reprogramming is over
Reboot the system
```

4.3 Initializing the System

Checking Power

If your application is a laptop or palmtop computer, first check the battery to make sure there is enough power to do the update. If not, inform the user to recharge the system before continuing the update and exit the update program. This ensures that the system won't stop in the middle of an update. Next, initialize access to flash for reads and writes, then try reading the device ID through the Command Register. Checking the device ID before programming or erasing helps determine if reads and writes work correctly and that the flash memory in the system matches your code before starting to reprogram the part. The manufacturer ID for Intel flash memories is **89H** (10001001), located at device address 00000H. Device IDs are located at address 00001H; the ID for the 28F001BX-T is **94H** (10010100), and the 28F001BX-B device ID is **95H** (10010101). These device addresses, in the DOS memory map, correspond to system addresses E0000H (mfgr. ID) and E0001H (device ID). If A_{16} inversion is used as described in Section 3.1, system addresses for mfgr. ID and device ID under normal operation are F0000H and F0001H.

NOTE:
During the initialization, you can also perform a scan of the adaptor space to ascertain if there is more flash in the system. Other Intel Flash Memories share common manufacturer IDs but have unique device IDs, listed below:

Device	Device ID (Hex)	Device ID (Binary)
28F256A	B9H	10111001
28F512	B8H	10111000
28F010	B4H	10110100
28F020	BDH	10111101
28F001BX-T	94H	10010100
28F001BX-B	95H	10010101

4.4 Code Loader Routine

The update utility described in the previous section provides an optional mouse-driven color graphical user interface (GUI) and allows not only BIOS update to the main block but also update of the parameter blocks, and copy/compare of block data to a DOS file. These types of features convey to the end user the ease and simplicity of performing a BIOS update. For example, the main block update utility lists all possible BIOS files in the selected drive and directory, and prompts the user for the desired file. System OEMs may want to encode a specific BIOS file name into the generic loader utility ".COM" or ".EXE" file. This allows automatic reading of the new BIOS file into a program buffer, bypassing the user prompt.

Once the file is loaded into RAM, the routine informs the user of the impending BIOS update and provides the option to exit if desired. If continued, it warns the user to not turn off power or reboot during the BIOS update procedure. It then erases and reprograms the main block with new BIOS data, notifies the user of successful update and reboots.

4.5 Flash Reprogramming Routines

On-Chip Erase Algorithm

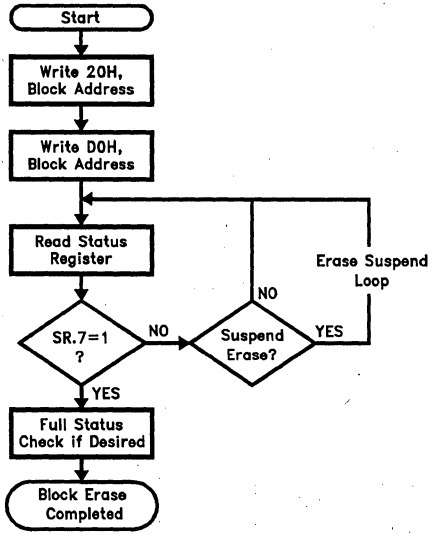
The 28F001BX system erase algorithm is shown in Figure 15. Note that the actual device erase algorithm (Quick-Erase) is controlled internally, including all timing and block preconditioning. This provides the same high level of reliability proven on Intel's ETOX II technology, while reducing system debug efforts. Erase progress is reported to system software thru specific Status Register bits. The 28F001BX erases all bits of a block in parallel. Minimum and typical erase times for each block are listed below:

Block	Minimum Time (Sec)	Typical Time (Sec)
Parameter (ea.)	1.3	2.1
Main	3.0	3.8
Boot	1.3	2.1

The actual erase time depends on the V_{PP} voltage level (11.4V–12.6V), temperature and the number of erase cycles already completed on the part. System software must comprehend adequate time for V_{PP}, after enabled, to ramp to 12V before erase is attempted. Capacitors on the V_{PP} bus, in addition to the intrinsic pump nature of many 12V solutions, cause an RC ramp. Systems that direct-wire 12V need not worry about this delay.

Erase Suspend/Resume

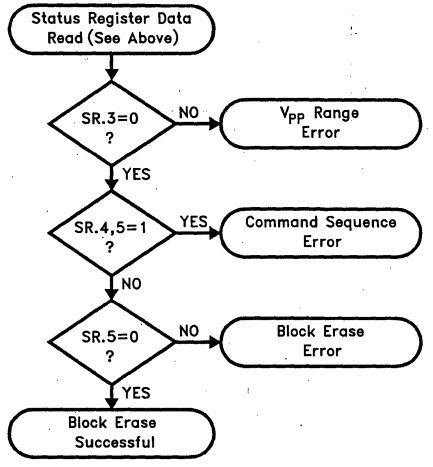
Erase suspend gives the user the ability, while erasing a block of the 28F001BX, to read data or execute code from another block. This capability, in conjunction with the minimal system overhead provided by the WSM, makes disabling of interrupts during block erase unnecessary. Once given the erase suspend command, the WSM halts, reports suspend status to the Status Register and allows array reads. When issued erase resume, it proceeds at the point where it was suspended. Figure 16 details the system code flowchart that suspends and resumes erase.



292077-14

Bus Operation	Command	Comments
Write	Erase Setup	Data = 20H Address = Within Block to be erased
Write	Erase	Data = D0H Address = Within Block to be erased
Read		Status Register Data. Toggle $\overline{OE}$ or $\overline{CE}$ to update Status Register
Standby		Check SR.7 1 = Ready, 0 = Busy
Repeat for subsequent blocks. Full status check can be done after each block or after a sequence of blocks. Write FFH after the last block erase operation to reset the device to Read Array Mode.		

FULL STATUS CHECK PROCEDURE

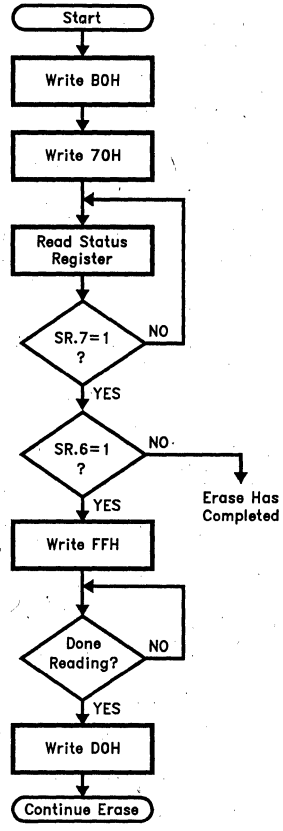


292077-15

Bus Operation	Command	Comments
Standby		Check SR.3 1 = V_{PP} Low Detect
Standby		Check SR.4, 5 Both 1 = Command Sequence Error
Standby		Check SR.5 1 = Block Erase Error
SR.3 MUST be cleared, if set during an erase attempt, before further attempts are allowed by the Write State Machine. SR.5 is only cleared by the Clear Status Register Command, in cases where multiple blocks are erased before full status is checked. If error is detected, clear the Status Register before attempting retry or other error recovery.		

Figure 15. 28F001BX Block Erase Algorithm

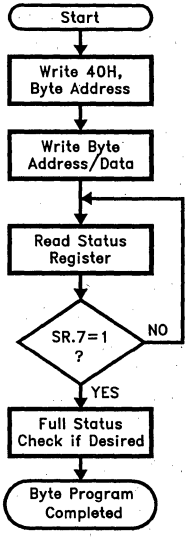
3



Bus Operation	Command	Comments
Write	Erase Suspend	Data = B0H
Write	Read Status Register	Data = 70H
Standby/ Read		Read Status Register Check SR.7 1 = Ready, 0 = Busy Toggle $\overline{OE}$ or $\overline{CE}$ to update Status Register
Standby		Check SR.6 1 = Suspended
Write	Read Array	Data = FFH
Read		Read array data from block other than that being erased.
Write	Erase Resume	Data = D0H

292077-16

Figure 16. 28F001BX Erase Suspend/Resume Algorithm



292077-17

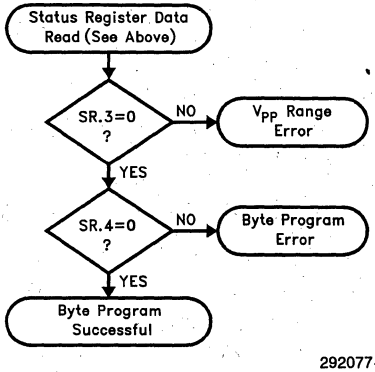
Bus Operation	Command	Comments
Write	Program Setup	Data = 40H Address = Byte to be programmed
Write	Program	Data to be Programmed Address = Byte to be programmed
Read		Status Register Data. Toggle $\overline{OE}$ or $\overline{CE}$ to update Status Register
Standby		Check SR.7 1 = Ready, 0 = Busy

Repeat for subsequent bytes.

Full status check can be done after each byte or after a sequence of bytes.

Write FFH after the last byte programming operation to reset the device to Read Array Mode.

FULL STATUS CHECK PROCEDURE



292077-18

Bus Operation	Command	Comments
Standby		Check SR.3 1 = V_{PP} Low Detect
Standby		Check SR.4 1 = Byte Program Error

SR.3 MUST be cleared, if set during a program attempt, before further attempts are allowed by the Write State Machine.

SR.4 is only cleared by the Clear Status Register Command, in cases where multiple bytes are programmed before full status is checked.

If error is detected, clear the Status Register before attempting retry or other error recovery.

Figure 17. 28F001BX Byte Programming Algorithm

On-Chip Programming Algorithm

As with 28F001BX erase, the Intel flash Quick-Pulse algorithm is internally controlled by the WSM. Figure 17 shows a system software flowchart for the Command Register/Status Register interface. Minimum and typical programming times (per byte) are 15 μs and 18 μs, respectively. Actual time varies with V_{PP}, temperature and cumulative programming cycles on the device. Ensure that stable 12V is applied to the device before attempting byte programming.

Full Status Checks

After polling the Status Register and determining that the WSM is again READY, system software should further analyze the Status Register to ensure that program or erase has successfully completed. The WSM will return to READY status after program or erase command sequences under any of the following conditions:

- Program/erase completed successfully,
- V_{PP} transition below specification during the program/erase attempt,
- Improper sequence of erase setup/confirm commands to the WSM, or
- Inability to erase the specified block, or program the desired byte.

Figures 15 and 17 detail the additional Status Register data analysis to ensure that program or erase have successfully occurred.

4.6 Recovery Routine Overview

Unsuccessful BIOS update can occur for any of the reasons listed below:

1. V_{PP} transitions out of specified tolerance during program or erase.
2. Incorrect code in the update BIOS disk file, or damaged BIOS disk.
3. Loss of system power during program or erase.
4. System reset (such as reboot) during program or erase.

The Status Register, through bit 3, reports V_{PPH} loss to system software. The BIOS update utility can detect scenario 1 and recover by simply re-attempting block update.

A checksum of update BIOS code after copy from disk to RAM, before flash erase and reprogram, will eliminate error caused by scenario 2.

PC motherboard logic should gate the 28F001BX $\overline{\text{PWD}}$ pin with both POWER GOOD and $\overline{\text{RESET}}$ signals, to abort program or erase attempts if either scenario 3 or 4 were to occur. This allows the processor to execute code out of the 8 KByte boot block upon system recovery. System reset or loss of system power will clear the Status Register to value 80H and leave the block being updated partially programmed or erased. As detailed previously in Section 3.1, a checksum of the main block will alert the system to an incomplete BIOS. Recovery is achieved by the following or similar steps:

- Initialize CPU and system logic.
- Initialize the system floppy disk.
- Prompt the user to insert a BIOS diskette, through speaker "beep".
- Erase and reprogram the main and/or parameter blocks with file data.
- Reboot

4.7 Power Management

Battery-powered PCs incorporate a variety of techniques to prolong system life between recharges. Typically, power management software senses user inactivity and shuts off power-intensive sections of the system. Options include:

- Display powerdown
- Disk/hard drive powerdown
- System clock slowdown or suspend, and
- Powerdown of non-volatile circuitry in the system.

The 28F001BX fits the latter description. When the $\overline{\text{PWD}}$ pin transitions to GND, the device enters an ultra-low power mode, typically consuming 0.25 μW thru V_{CC}. This technique can also be used to power down the BIOS memory after BIOS code has been shadowed to DRAM, if available in the system. When not programming or erasing the 28F001BX, the system should shut off 12V V_{PP} to the part to minimize current draw through this supply.

User inactivity is typically detected if the keyboard has not been used, or the disk drive has not been accessed, for a predetermined interval (this is often user-programmable). Power management software must ensure that a BIOS update is not occurring, before powering down the 28F001BX, to prevent incomplete update.

For more information on power management techniques, consult datasheets and application notes on the Intel386SL microprocessor superset.

5.0 SUMMARY

5.1 Traditional BIOS Storage and Disadvantages

Traditional BIOS storage has been in EPROM, which offers nonvolatility and factory programming capability. In earlier PCs, the BIOS code was fairly simple (relative to today's software) and updates were infrequent, so EPROMs or ROMs were an acceptable BIOS storage medium. Today's systems are much more sophisticated, with many designs supporting the Intel i386/i486™ microprocessors and new bus architectures like MCA and EISA for the first time. These new buses allow peripherals to take control of the system bus . . . it is difficult to guess what new system configurations might emerge. Therefore, the potential for a change in the BIOS code is much greater and the frequency of change is likely to increase.

A system designer may use EPROMs for BIOS storage to reduce initial system (component) costs, but the long-term update cost is much more than the difference between EPROM and flash memory components. A major manufacturer of PCs has estimated that a service call for a BIOS update with EPROMs can cost upwards of $300.00 for **ONE** update at **ONE** site. EPROMs are also susceptible to bent leads during insertion by the technician, or more likely, the end user. Service is becoming a key differentiator between the multitudes of PC makers. Reducing the number of times a PC has to be opened for any reason and providing improved service increases customer confidence and promotes a reliable image.

5.2 Advantages of an Updatable BIOS

Using flash memory for BIOS storage provides a flexible code medium that allows the BIOS code to adapt to changing hardware and software conditions. BIOS updates in flash are inexpensive, via a floppy disk or modem. They remove EPROM inventories, reduce packaging requirements, reduce total postage costs and eliminate service cost for BIOS code updates by removing the need for a technician to do the update. A company that supports mutiple OEMs can improve version management control by using a flash BIOS and floppies or a BBS for updates. An additional benefit is that not only the BIOS, but DOS itself can be stored in the same flash memory device.

5.3 Advantages of Adding DOS in FLASH

Once the requirements for flash memory BIOS are met, the capability is also in place for adding DOS in FLASH. Why put DOS in FLASH? For laptop and palmtop PCs, battery longevity is of paramount concern, followed closely by weight and increasing user RAM (640 KB) space. Extra user RAM is needed for applications that require more than the typical 570 KBytes (640 KB–70 KB) available with disk-based DOS. Digital Research Incorporated and Microsoft both make "DOS-in-ROM" products that address these needs. MS-DOS ROM version 3.22 is an example.

Microsoft's MS-DOS ROM Version 3.22 is a full-function version of MS-DOS 3.2. It features instant-on and employs only 15 KB of the 640 KB DOS RAM user space, leaving the rest for applications. Since MS-DOS ROM Version 3.22 loads from adaptor space, both disk access and DOS loadtime are reduced. For laptops, anything that can reduce disk access equates to battery longevity. Laptops can reduce weight by using MS-DOS ROM Version 3.22 and replacing the floppy drive with an IC card. Adding MS-DOS ROM Version to desktops also liberates additional user RAM for the same above reasons, but may not be optimal for high speed 32-bit systems.

All future versions of MS-DOS will be supported with equivalent versions of MS-DOS ROM. See Appendix B for more information.

5.4 Advantages of Adding 1 MB–4 MB of Resident Code Storage

There is a growing need for systems to be able to provide a small suite of bundled applications. Benefits to the user are faster application execution thru reduced hard or floppy disk access, no power used to store the resident code, and instant-on. No time is wasted transferring data over a disk I/O interface. The code is instead loaded to RAM with a simple memory copy function or procedure. In some cases, code is directly executed by the processor. Tandy's Deskmate is an example of such a system. Future versions of Deskmate-like user interfaces could easily be made flash-updatable. SRAM is too expensive and requires power to just store files. Furthermore, battery backup is not a reliable means of achieving nonvolatility. Intel's Flash Memory can provide user configurability for 1 MB–4 MB of code storage for just 2x–3x the cost of EPROMs and less than half the cost of SRAM. Applications such as Lotus 123, WordPerfect and Microsoft Works also come in either a direct-execute "ROM" version or a load-from-ROM format. Many other ROM application software packages are in development, servicing the successful and growing needs of the laptop/palmtop computers. Therefore, if an application can be stored or runs from ROM, it can be stored and run from flash. As software packages are periodically updated, flash memory provides the capability of updating these "ROM" applications at little cost to the software vendor and with no system disassembly required.

APPENDIX A
SOFTWARE ROUTINES

```
/*****************************************************************************/
/*  Copyright Intel Corporation, 1991                                        */
/*  Brian Dipert, Intel Corporation, July 14, 1991, Revision 1.4             */
/*  The following drivers control the Command and Status Registers of        */
/*    the 28F001BX Flash Memory to drive byte program, block erase, Status   */
/*    Register read and clear and array read algorithms.                     */
/*    Sample Vpp and /PWD control blocks are also included.                  */
/*  The functions listed below are included:                                 */
/*    erasbgn(): Begins block erasure                                        */
/*    erassusp(): Suspends erase to allow reading data from a block of the   */
/*                28F001BX other than that being erased                      */
/*    erasres(): Resumes erase if suspended                                  */
/*    end():     Polls the Write State Machine to determine if block erase or*/
/*               byte program have completed                                 */
/*    eraschk(): Executes full status check after erase completion           */
/*    progbgn(): Begins byte programming                                     */
/*    progchk(): Executes full status check after byte program completion    */
/*    idread():  Reads and returns the manufacturer and device IDs of the    */
/*               target 28F001BX                                             */
/*    statrd():  Reads and returns the contents of the Status Register       */
/*    statclr(): Clears the Status Register                                  */
/*    rdmode():  Puts the 28F001BX in Read Array mode                        */
/*    rdbyte():  Reads and returns a specified byte from the target 28F001BX */
/*    vppup():   Enables high voltage Vpph                                   */
/*    vppdown(): Disables Vpph                                               */
/*    pwdon():   Ramps the /PWD pin to high voltage Vhh, enabling boot block */
/*               program/erase                                               */
/*    pwdoff():  Disables high voltage Vhh on /PWD, disabling program        */
/*               and erase of boot block                                     */
/*                                                                           */
/*  Addresses are transferred to functions as pointers to far bytes (ie long */
/*    integers). An alternate approach is to create a global array the size of the */
/*    28F001BX and locate "over" the 28F001BX in the system memory map. Accessing */
/*    specific locations of the 28F001BX is then accomplished by passing the chosen */
/*    function an offset from the array base versus a specific address. Different */
/*    microprocessor architectures will require different array definitions; ie for */
/*    the x86 architecture, define it as "byte boot [2][10000]" and pass each */
/*    function TWO offsets to access a specific location. MCS-51 architectures */
/*    are limited to "byte boot[10000]"; alternate approaches such as writing to */
/*    control bits will be required to access the full flash array           */
/*                                                                           */
/* To create a far pointer, a function such as MK_FP() can be used, given    */
/*    a segment and offset in the x86 architecture. I use Turbo-C; see your  */
/*    compiler reference manual for additional information.                  */
/*****************************************************************************/
```

```
/*****************************************************************************/
/*  Revision History: Rev 1.4                                                */
/*                                                                           */
/*  Changes from 1.0 to 1.1: Added typedef for "byte" to accurately reflect  */
/*    this x8 device. Altered variable definitions accordingly. Combined     */
/*    functions progend() and erasend() into function end().                 */
/*                                                                           */
/*  Changes from 1.1 to 1.2: Added this revision history block. Added above  */
/*    comments on alternate addressing methods.                              */
/*                                                                           */
/*  Changes from 1.2 to 1.3: Added pass/fail error return from idread(),     */
/*    idread() at beginning of progbgn() and erasbgn(), pass/fail error      */
/*    return from progbgn() and erasbgn().                                   */
/*                                                                           */
/*  Changes from 1.3 to 1.4: Revised code to reflect simplified program and  */
/*    erase algorithms. 28F001BX automatically transitions to Read Status Register */
/*    mode after program command sequence, erase command sequence and remains in */
/*    Read Status Register mode after Erase Suspend is issued. Address 0000H is no */
/*    longer required to read or clear the Status Register.                  */
/*****************************************************************************/

typedef unsigned char byte;

/*****************************************************************************/
/* Function: Main                                                            */
/* Description: Included only to omit errors when attempting to compile code. */
/*  The end customer would insert their main program here.                   */
/*****************************************************************************/

main()
{
}

/*****************************************************************************/
/*  Function: Erasgbn                                                        */
/*    Description: Begins erase of a block.                                  */
/*    Inputs: blckaddr: System address within the block to be erased         */
/*    Outputs: None                                                          */
/*    Returns: 0 = Erase successfully initiated                              */
/*             1 = Erase not initiated (ID check error)                      */
/*    Device Read Mode on Return: Status Register (ID if returns 1)          */
/*****************************************************************************/

#define ERASETUP    0X20    /* Erase Setup command                           */
#define ERASCONF    0XD0    /* Erase Confirm command                         */

int erasbgn(blckaddr)

byte far *blckaddr;         /* blckaddr is an address within the block to be */
                            /*    erased                                     */

{
 if (idread()==1)           /* ID read error; device not powered up?         */
     return (1);
 *blckaddr = ERASETUP;      /* Write Erase Setup command to block address    */
 *blckaddr = ERASCONF;      /* Write Erase Confirm command to block address  */
 return (0);
}
```

```
/*****************************************************************************/
/*  Function: Erassusp                                                       */
/*    Description: Suspends block erase to read from another block           */
/*    Inputs: None                                                           */
/*    Outputs: None                                                          */
/*    Returns: 0 = Erase suspended                                           */
/*             1 = Error; Write State Machine not busy (erase suspend not possible) */
/*    Device Read Mode on Return: Read Status Register                       */
/*****************************************************************************/

#define RDYMASK     0X80    /* Mask to isolate the WSM Status bit of the      */
                            /*      Status Register                           */
#define WSMRDY      0X80    /* Status Register value after masking, signifying */
                            /*      that the WSM is no longer busy            */
#define SUSPMASK    0X40    /*Mask to isolate the Erase Suspend Status bit of the */
                            /*      Status Register                           */
#define ESUSPYES    0X40    /* Status Register value after masking, signifying */
                            /*      that erase has been suspended             */
#define STATREAD    0X70    /* Read Status Register command                   */
#define SYSADDR     0       /* This constant can be initialized to any address */
                            /*      within the memory map of the target 28F001BX */
                            /*      and is alterable depending on the system  */
                            /*      architecture                              */
#define SUSPCMD     0XB0    /* Erase Suspend command                          */

int erassusp()

{
 byte far *stataddr;        /* Pointer variable used to write commands to device */

 stataddr  = (byte far *)SYSADDR;
 *stataddr = SUSPCMD;       /* Write Erase Suspend command to the device      */
 *stataddr = STATREAD;      /* Write Read Status Register command to 28F001BX */
 while ((*stataddr & RDYMASK)  != WSMRDY)
    ;                       /* Will remain in while loop until bit 7 of the   */
                            /*      Status Register goes to 1, signifying that the */
                            /*      WSM is no longer busy                     */
 if ((*stataddr & SUSPMASK) == ESUSPYES)
            return(0);      /* Erase is suspended ... return code "0"         */
 return(1);                 /* Erase has already completed; suspend not possible. */
                            /*      Error code "1"                            */
}
```

```
/*****************************************************************************/
/*  Function: Erasres                                                        */
/*    Description: Resumes block erase previously suspended                  */
/*    Inputs: None                                                           */
/*    Outputs: None                                                          */
/*    Returns: 0 = Erase resumed                                             */
/*             1 = Error; Erase not suspended when function called           */
/*    Device Read Mode on Return: Status Register                            */
/*****************************************************************************/

#define RDYMASK     0X80    /* Mask to isolate the WSM Status bit of the     */
                            /*      Status Register                          */
#define WSMRDY      0X80    /* Status Register value after masking, signifying */
                            /*      that the WSM is no longer busy           */
#define SUSPMASK    0X40    /* Mask to isolate the Erase Suspend Status bit  */
                            /*      of the Status Register                   */
#define ESUSPYES    0X40    /* Status Register value after masking, signifying */
                            /*      that erase has been suspended            */
#define STATREAD    0X70    /* Read Status Register Command                  */
#define SYSADDR     0       /* This constant can be initialized to any       */
                            /*      address within the memory map of the target */
                            /*      28F001BX and is alterable depending on the */
                            /*      system architecture                      */
#define RESUMCMD    0XD0    /* Erase Resume Command                          */

int erasres()

{
byte far *stataddr;         /* Pointer variable used to write commands to device */

stataddr  = (byte far *)SYSADDR,
*stataddr = STATREAD;       /* Write Read Status Register command to 28F001BX */
if ((*stataddr & SUSPMASK)  != ESUSPYES)
    return (1);             /* Erase not suspended. Error code "1"           */
*stataddr = RESUMCMD;       /* Write Erase Resume command to the device      */
while ((*stataddr & SUSPMASK) == ESUSPYES)
    ;                       /* Will remain in while loop until bit 6 of the  */
                            /*      Status Register goes to 0, signifying    */
                            /*      erase resumption                         */
while ((*stataddr & RDYMASK) == WSMRDY)
    ;                       /* Will remain in while loop until bit 7 of the  */
                            /*      Status Register goes to 0, signifying    */
                            /*      that the WSM is once again busy          */
return (0);
}
```

```
/*****************************************************************************/
/*  Function: End                                                            */
/*    Description: Checks to see if the WSM is busy                          */
/*                 (is program/erase completed?)                             */
/*    Inputs: None                                                           */
/*    Outputs: statdata: Status Register data read from device               */
/*    Returns: 0 = Program/Erase completed                                   */
/*             1 = Program/Erase still in progress                           */
/*    Device Read Mode on Return: Status Register                            */
/*****************************************************************************/

#define RDYMASK     0X80    /* Mask to isolate the WSM Status bit of the     */
                            /*      Status Register                          */
#define WSMRDY      0X80    /* Status Register value after masking, signifying */
                            /*      that the WSM is no longer busy           */
#define STATREAD    0X70    /* Read Status Register command                  */
#define SYSADDR     0       /* This constant can be initialized to any       */
                            /*      address within the memory map of the target */
                            /*      28F001BX and is alterable depending on the */
                            /*      system architecture                      */

int end (statdata)

byte *statdata;             /* Allows Status Register data to be passed back */
                            /*      to the main program for further analysis */
{
 byte far *stataddr;        /* Pointer variable used to write commands to    */
                            /*      device                                   */
 stataddr  = (byte far*)SYSADDR;
 *stataddr = STATREAD;      /* Write Read Status Register command to 28F001BX */
 if (((*statdata = *stataddr) & RDYMASK)  != WSMRDY)
     return (1);            /* Program/erasure still in progress...code "1"  */
 return (0);                /* Program/erase attempt completed...code "0"    */
}
```

```
/*****************************************************************************/
/*  Function: Eraschk                                                        */
/*    Description: Completes full Status Register check for erase (proper    */
/*      command sequence, VPP low detect, erase success). This routine assumes*/
/*      that erase completion has already been checked in function end() and  */
/*      therefore does not check the WSM Status bit of the Status Register    */
/*    Inputs: statdata: Status Register data read in function end             */
/*    Outputs: None                                                          */
/*    Returns: 0 = Erase completed successfully                              */
/*             1 = Error; Vpp low detect                                     */
/*             2 = Error; Block erase error                                  */
/*             3 = Error; Improper command sequencing                        */
/*    Device Read Mode on Return: Same as when entered                        */
/*****************************************************************************/

#define ESEQMASK    0X30    /* Mask to isolate the Erase and Program          */
                            /*         Status bits of the Status Register     */
#define ESEQFAIL    0X30    /* Status Register value after masking if erase   */
                            /*         command sequence error has been detected*/
#define ERRMSK      0X20    /* Mask to isolate the Erase Status bit of the    */
                            /*         Status Register                        */
#define ERASERR     0X20    /* Status Register value after masking if erase error */
                            /*         has been detected                      */
#define VLOWMASK    0X08    /* Mask to isolate the Vpp Status bit of the Status */
                            /*         Register                               */
#define VPPLOW      0X08    /* Status Register value after masking if Vpp low */
                            /*         has been detected                      */

int eraschk(statdata)

 byte statdata;             /* Status Register data that has been already read */
                            /*         from the 28F001BX in function end()    */

{
 if ((statdata & VLOWMASK) == VPPLOW)
      return (1);           /* Vpp low detect error, return code "1"          */
 if ((statdata & ERRMSK) == ERASERR)
      return (2);           /* Block erase error detect, return code "2"      */
 if ((statdata & ESEQMASK) == ESEQFAIL)
      return (3);           /* Erase command sequence error, return code "3"  */
 return (0);                /* Block erase success, return code "0"           */
}
```

```
/*****************************************************************************/
/*  Function: Progbgn                                                        */
/*    Description: Begins byte program sequence                              */
/*    Inputs: pdata: Data to be programmed into the device                   */
/*            paddr: Target address to be programmed                         */
/*    Outputs: None                                                          */
/*    Returns: 0 = Program successfully initiated                            */
/*             1 = Program not initiated (ID check error)                    */
/*    Device Read Mode on Return: Status Register (ID if returns 1)          */
/*****************************************************************************/

#define SETUPCMD    0X40    /*Program Setup command                          */

int progbgn (pdata,paddr)

byte pdata;                 /* Data to be programmed into the 28F001BX       */
byte far *paddr;            /* paddr is the destination address for the data */
                            /*         to be programmed                      */

{
 if (idread() == 1)         /* Device ID read error...powered up?            */
     return (1);
*paddr = SETUPCMD;          /* Write Program Setup command and               */
                            /*         destination address                   */
*paddr = pdata;             /* Write program data                            */
                            /*         and destination address               */
 return (0);
}
```

```
/*****************************************************************************/
/*  Function: Progchk                                                        */
/*   Description: Completes full Status Register check for byte program (VPP low  */
/*     detect, programming success). This routine assumes that byte program   */
/*     completion has already been checked in function end() and              */
/*     therefore does not check the WSM Status bit of the Status Register     */
/*   Inputs: statdata: Status Register data read in function end              */
/*   Outputs: None                                                            */
/*   Returns: 0 = Byte programming completed successfully                     */
/*            1 = Error; VPP low detect                                       */
/*            2 = Error; Byte program error                                   */
/*   Device Read Mode on Return: Status Register                              */
/*****************************************************************************/

#define PERRMSK   0X10  /* Mask to isolate the Program Status bit of the     */
                        /*            Status Register                        */
#define PROGERR   0X10  /* Status Register value after masking if program    */
                        /*            error has been detected                */
#define VLOWMASK  0X08  /* Mask to isolate the VPP Status bit of the Status  */
                        /*            Register                               */
#define VPPLOW    0X08  /* Status Register value after masking if VPP low    */
                        /*            has been detected                      */

int progchk (statdata)

byte statdata;          /* Status Register data that has been already read   */
                        /*            from the 28F001BX in function end()    */

{
 if ((statdata & VLOWMASK) == VPPLOW)
      return (1);       /* VPP low detect error, return code "1"             */
 if ((statdata & PERRMSK) == PROGERR)
      return (2);       /* Byte program error detect, return code "2"        */
 return (0);            /* Byte/string program success, return code "0"      */
}
```

```
/*****************************************************************************/
/*  Function: Idread                                                         */
/*    Description: Reads the manufacturer and device IDs from the target 28F001BX  */
/*    Inputs: None                                                           */
/*    Outputs: mfgrid: Returned manufacturer ID                              */
/*             deviceid: Returned device ID                                  */
/*    Returns: 0 = ID read correct                                           */
/*             1 = Wrong or no ID                                            */
/*    Device Read Mode on Return: inteligent Identifier                      */
/*****************************************************************************/

#define MFGRADDR      0         /* Address "0" for the target 28F001BX...    */
                                /*         alterable depending on the system */
                                /*         architecture                      */
#define DEVICADD      1         /* Address "1" for the target 28F001BX...    */
                                /*         alterable depending on the system */
                                /*         architecture                      */
#define IDRDCOMM      0X90      /* inteligent Identifier command             */
#define INTELID       0X89      /* Manufacturer ID for Intel devices         */
#define DVCIDBT       0X94      /* Device ID for 28F001BX-T; change to 95H if */
                                /*         using 28F001BX-B!!!               */

int idread(mfgrid,deviceid)

byte *mfgrid;                   /* The manufacturer ID read by this function, to */
                                /*         be transferred back to the calling */
                                /*         program                           */
byte *deviceid;                 /* The device ID read by this function, to be */
                                /*         transferred back to the calling function */

{
 byte far *tempaddr;            /* Pointer address variable used to read IDs */

 tempaddr = (byte far*)MFGRADDR;
 *tempaddr= IDRDCOMM;           /* Write inteligent Identifier command to an */
                                /*         address within the 28F001BX memory map */
                                /*         (in this case, 00H)               */
 *mfgrid  = *tempaddr;          /* Read mfgr ID, tempaddr still points at address "0" */
 tempaddr = (byte far*)DEVICADD;/* Point to address "1" for the device specific ID */
 *deviceid= *tempaddr;          /* Read device ID                            */
 if ((*mfgrid != INTELID)||(*deviceid != DVCIDBT))
     return (1);                /* ID read error; device powered up?         */
 return (0);
}
```

```
/*****************************************************************************/
/*  Function: Statrd                                                         */
/*     Description: Returns contents of the target 28F001BX Status Register  */
/*     Inputs: None                                                          */
/*     Outputs: statdata: Returned Status Register data                      */
/*     Returns: Nothing                                                      */
/*     Device Read Mode on Return: Status Register                           */
/*****************************************************************************/

#define STATREAD    0X70     /* Read Status Register command                 */
#define SYSADDR     0        /* This constant can be initialized             */
                             /*         to any address within the            */
                             /*         memory map of the target 28F001BX    */
                             /*         and is alterable depending on        */
                             /*         the system architecture              */
int statrd(statdata)

byte *statdata;              /* Allows Status Register data to               */
                             /*         be passed back to the calling program*/
                             /*         for further analysis                 */

{
 byte far *stataddr;         /* Pointer variable used to write               */
                             /*         commands to device                   */
 stataddr   = (byte far*)SYSADDR;
 *stataddr  = STATREAD;      /* Write Read Status Register                   */
                             /*         command to 28F001BX                  */
 *statdata  = *stataddr;
 return;
}
```

```
/*****************************************************************************/
/*  Function: Statclr                                                        */
/*    Description: Clears the 28F001BX Status Register                       */
/*    Inputs: None                                                           */
/*    Outputs: None                                                          */
/*    Returns: Nothing                                                       */
/*    Device Read Mode on Return: Status Register                            */
/*****************************************************************************/

#define STATCLER    0X50    /* Clear Status Register command                  */
#define SYSADDR     0       /* This constant can be initialized to any        */
                            /*         address within the memory map of the target*/
                            /*         28F001BX and is alterable depending on */
                            /*         the system architecture                */

int statclr()

{
 byte far *stataddr;        /* Pointer variable used to write commands to     */
                            /*         device                                 */

 stataddr  = (byte far*)SYSADDR;
 *stataddr = STATCLER;      /* Write Clear Status Register command to         */
                            /*         28F001BX                               */
 return;
}


/*****************************************************************************/
/*  Function: Rdmode                                                         */
/*    Description: Puts the target 28F001BX in Read Array Mode. This function*/
/*      might be used, for example, to prepare the system for return to code */
/*      execution out of the Flash memory after program or erase algorithms  */
/*      have been executed off-chip                                          */
/*    Inputs: None                                                           */
/*    Outputs: None                                                          */
/*    Returns: Nothing                                                       */
/*    Device Read Mode on Return: Array                                      */
/*****************************************************************************/

#define RDARRAY     0XFF    /* Read Array command                             */
#define SYSADDR     0       /* This constant can be initialized to any        */
                            /*       address within the memory map of the target */
                            /*       28F001BX and is alterable depending on */
                            /*       the system architecture                */

int rdmode()

{
 byte far *tempaddr;        /* Pointer variable used to write commands to     */
                            /*         device                                 */
 tempaddr  = (byte far*)SYSADDR;
 *tempaddr = RDARRAY;       /* Write Read Array command to 28F001BX           */
 return;
}
```

```
/*****************************************************************************/
/*  Function: Rdbyte                                                         */
/*    Description: Reads a byte of data from a specified address and         */
/*                 returns it to the calling program                         */
/*    Inputs: raddr: Target addres to be read from                           */
/*    Outputs: rdata: Data at the specified address                          */
/*    Returns: Nothing                                                       */
/*    Device Read Mode on Return: Array                                      */
/*****************************************************************************/

#define RDARRAY     0XFF    /* Read array command                            */

int rdbyte (rdata,raddr)

byte *rdata;                /* Returns data read from the device at          */
                            /*         specified address                     */
byte far *raddr;            /* Raddr is the target address to be read from   */

{
 *raddr = RDARRAY;          /* Write read array command to an address within */
                            /*         the 28F001BX memory map (in this case the */
                            /*         target address)                       */
 *rdata = *raddr;           /* Read from the specified address and store     */
 return;
}
```

```
/*****************************************************************************/
/*  Function: Vppup                                                          */
/*    Description: Ramps the Vpp supply to the target 28F001BX to enable     */
/*     programming or erase. This routine can be tailored to the individual  */
/*     system architecture. For purposes of this example, I assumed that a   */
/*     system Control Register existed at system address 20000 hex.          */
/*     with the following definitions:                                       */
/*                                                                           */
/*      Bit 7: Vpph Control:     1 = Enabled                                 */
/*                               0 = Disabled                                */
/*      Bit 6: PWD Control:      1 = PowerDown Enabled                       */
/*                               0 = PowerDown Disabled                      */
/*      Bits 5-0: Undefined                                                  */
/*    Inputs: None                                                           */
/*    Outputs: None                                                          */
/*    Returns: Nothing                                                       */
/*    Device Read Mode on Return: As existed before entering the function.   */
/*     Part is now ready for program or erase command sequence               */
/*****************************************************************************/

#define VPPHIGH              0X80     /* Bit 7 = 1, Vpp elevated to Vpph      */
#define SYSCADDR             0X20000  /* Assumed system Control Register Address */

int vppup()

{
 byte far *contaddr;                  /* Pointer variable used to write data  */
                                      /*         to the System Control Register */
 contaddr  = (byte far*)SYSCADDR;
 *contaddr = *contaddr | VPPHIGH;     /* Read current Control Register data,  */
                                      /*         "OR" with constant to ramp Vpp */
 return;
}
```

```
/*****************************************************************************/
/*  Function: Vppdown                                                        */
/*    Description: Ramps down the Vpp supply to the target 28F001BX to       */
/*     disable programming/erase. See above for a description of the         */
/*     assumed system Control Register.                                      */
/*    Inputs: None                                                           */
/*    Outputs: None                                                          */
/*    Returns: Nothing                                                       */
/*    Device Read Mode on Return: As existed before entering the function. Part */
/*     now has high Vpp disabled. If program or erase was in progress when   */
/*     this function was called, it will complete unsuccessfully with Vpp low error */
/*     in the Status Register.                                               */
/*****************************************************************************/

#define VPPDWN      0X7F    /* Bit 7 = 0, Vpp lowered to Vppl                 */

#define SYSCADDR    0X20000 /* Assumed system Control Register Address       */

int vppdown()

{
 byte far *contaddr;        /* Pointer variable used to write data to the    */
                            /*         system Control Register               */
 contaddr  = (byte far*)SYSCADDR;

 *contaddr = *contaddr & VPPDWN;
                            /* Read current Control Register data, "AND" with */
                            /*         constant to lower Vpp                 */
 return;
}
```

```
/*****************************************************************************/
/*  Function: Pwdon                                                          */
/*    Description: Toggles the 28F001BX /PWD pin low to put the device in Deep*/
/*     PowerDown mode. See above for a description of the assumed            */
/*     system Control Register.                                              */
/*    Inputs: None                                                           */
/*    Outputs: None                                                          */
/*    Returns: Nothing                                                       */
/*    Device Read Mode on Return: The part is powered down. If program or erase*/
/*     was in progress when this function was called, it will abort with     */
/*     resulting partially programmed or erased data. Recovery in the form of*/
/*     repeat of program or erase will be required once the part             */
/*     transitions out of powerdown, to initialize data to a known state.    */
/*****************************************************************************/

#define PWD        0X40      /* Bit 6 = 1, /PWD enable                       */
#define SYSCADDR   0X20000   /* Assumed system Control Register Address      */

int pwdon()

{
 byte far *contaddr;         /* Pointer variable used to write data to the   */
                             /*         system Control Register              */
 contaddr  = (byte far*)SYSCADDR;
 *contaddr = *contaddr | PWD;/* Read current Control Register data, "OR" with*/
                             /*         constant to enable Deep PowerDown    */
 return;
}
```

```
/*****************************************************************************/
/*  Function: Pwdoff                                                         */
/*    Description: Toggles the 28F001BX /PWD pin high to transition the part */
/*     out of Deep PowerDown. See above for a description of the assumed system */
/*     Control Register.                                                     */
/*    Inputs: None                                                           */
/*    Outputs: None                                                          */
/*    Returns: Nothing                                                       */
/*    Device Read Mode on Return: Read Array mode. Low voltage is removed    */
/*     from /PWD. 28F001BX output pins will output valid data time tPHQV     */
/*     after the /PWD pin transitions high (reference the datasheet AC       */
/*     Read Characteristics) assuming valid states on all other control      */
/*     and power supply pins.                                                */
/*****************************************************************************/

#define PWDOFF        0XBF       /* Bit 6 = 0, /PWD disabled                 */
#define SYSCADDR      0X20000    /* Assumed system Control Register Address  */

int pwdoff()

{
 byte far *contaddr;             /* Pointer variable used to write data to the */
                                 /*         system Control Register          */
 contaddr  = (byte far*)SYSCADDR;
 *contaddr = *contaddr & PWDOFF; /* Read current Control Register data, "AND" with */
                                 /*         constant to disable Deep PowerDown */
 return;
}
```

3

APPENDIX B
MS-DOS ROM VERSION OVERVIEW

Technical Highlights

(Taken from Microsoft Product Overview)

RAM Economy

Because MS-DOS ROM Version executes from ROM, only 15 KB of system RAM space is required for MS-DOS. For a typical user, this will result in a savings of about 40 KB of RAM over disk-based MS-DOS. As a result of this savings, the user is able to run more programs and work with larger data files with the ROM Version than with disk-based MS-DOS. Instant-On MS-DOS ROM Version provides a significant reduction in "boot time", or the amount of time it takes from the completion of the power-on self test until a DOS prompt appears. With the ROM Version, this typically takes one second.

No End-User Installation

MS-DOS ROM Version is pre-installed by the OEM (original equipment manufacturer) in the system, thus freeing end users from the task of installing MS-DOS.

Adaptable to OEM Hardware Platforms

MS-DOS ROM Version is structured such that it allows the OEM to include a specific routine to determine which drive to boot from and any specific parameters if booting from the ROM drive. This makes it possible to easily port the ROM Version to a wide variety of hardware environments. MS-DOS ROM Version is also positioned independent, in that it can reside anywhere in the "reserved" space (the area between 640 KB and 1 MB). This provides an additional Version to the specific requirements of the OEM's hardware platform.

ROM Economy

MS-DOS ROM Version occupies only 62 KB of ROM space, thus minimizing the amount of ROM that an OEM must include in the system. Three modules reside in the reserved space: COMMAND.COM, IO.SYS and the DOS Kernel. All three are position independent, so an OEM can decide where to place these modules in the reserved area.

National Language Support

Microsoft offers a full compliment of localized version of MS-DOS ROM Version, including Kanji and Chinese translations.

Ease of Development

As PCs become the engines for many embedded applications, manufacturers would like to develop new applications utilizing existing PC software tools. MS-DOS ROM allows manufacturers to take full advantage of these tools. For instance, a programmer can develop and debug an application onto a PC subsystem which may be embedded into a larger system. This benefit translates into a cost savings when developing a solution for vertical markets.

APPENDIX C
BIOS VENDOR INFORMATION

American Megatrends Inc. (AMI)
1346 Oakbrook Drive, Suite 120
Norcross, GA 30093
(404) 263-8181

Award Software Inc.
130 Knowles Drive
Los Gatos, CA 95030
(408) 370-7979

Phoenix Technologies, LTD.
40 Airport Parkway
San Jose, CA 95110
(408) 452-6500

Systemsoft Corporation
313 Speen Street
Natick, MA 01760
(508) 651-0088

This list is intended for example only, and in no way represents all companies that support BIOS software. Since this industry developes many new solutions each year, Intel recommends that the designer contact the vendors for their latest products. Intel will continue to work with BIOS vendors to develop optimum solutions. Intel Corporation assumes no responsibility for circuitry or software other than circuitry embodied in Intel products. No software patent licenses are implied.

APPENDIX D
MICROPROCESSOR/MICROCONTROLLER COMPATIBILITY CHART

28F001BX-T	28F001BX-B
x86 Family	i960™ KA/KB Microprocessor
i860™ Family	i960™ SA/SB Microprocessor
i960™ CA Microprocessor	MCS®-51 Family
	MCS®-96 Family

REVISION HISTORY

Number	Description
-004	Added 10K resistor to FET output, Figure 13. Updated Erase Suspend Flowchart, Figure 16 Updated Erase Suspend "C" Code, page A-3